МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Построение и анализ алгоритмов»

Тема: Поиск с возвратом

Студент гр. 0304 _____ Алексеев Р.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2022

Цель работы.

Изучить алгоритм поиска с возвратом (бэктрекинг) на примере задачи разбиения квадрата на минимальное количество квадратов меньшего размера. Изучить рекурсивный способ реализации алгоритма.

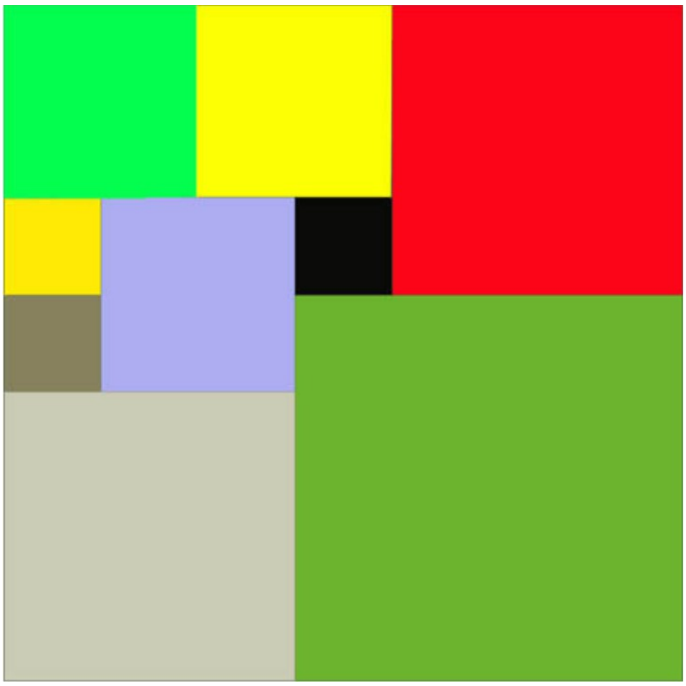
Задание.

Вариант 2р

Рекурсивный бэктрекинг. Исследование времени выполнения от размера квадрата.

У Вовы много квадратных обрезков доски. Их стороны (размер) изменяются от 1 до N - 1, и у него есть неограниченное число обрезков любого размера. Но ему очень хочется получить большую столешницу - квадрат размера N. Он может получить ее, собрав из уже имеющихся обрезков(квадратов).

Например, столешница размера 7×7 может быть построена из 9 обрезков.



Внутри столешницы не должно быть пустот, обрезки не должны выходить за пределы столешницы и не должны перекрываться. Кроме того, Вова хочет использовать минимально возможное число обрезков.

Входные данные

Размер столешницы - одно целое число N ($2 \le N \le 20$).

Выходные данные

Одно число K, задающее минимальное количество обрезков(квадратов), из которых можно построить столешницу(квадрат) заданного размера N. Далее должны идти K строк, каждая из которых должна содержать три целых числа x, y, и w, задающие координаты левого верхнего угла ($1 \le x,y \le N$) и длину стороны соответствующего обрезка(квадрата).

Пример входных данных

7

Соответствующие выходные данные

9

1 1 2

1 3 2

3 1 1

4 1 1

3 2 2

5 1 3

4 4 4

1 5 3

3 4 1

Выполнение работы.

Описание алгоритма

Задача разбиения квадрата на квадраты меньшего размера решается с помощью рекурсивного алгоритма бэктрекинга. На каждом шаге на битовой матрицы заполнения ищется пустая клетка, чтобы в неё вставить новый квадрат. Также при помощи битовой матрицы можно проверить, возможна ли вставка квадрата определенного размера по заданным координатам.

Суть алгоритма заключается в сравнении на каждом шаге количества квадратов в текущем варианте с количеством в минимальном варианте, если оно больше или равно, то вариант обработка варианта прекращается, т. к. он не может быть минимальным, если количество меньше, то запускается новый шаг алгоритма. Если все пустые клетки были заполнены и количество квадратов оказалось меньше чем в минимальном варианте, то текущий вариант сохраняется как новый минимальный, дальнейшее сравнение будет производиться уже с ним.

Для добавления нового квадрата сначала находится верхняя левая пустая клетка, после чего в неё поочередно вставляются квадраты размеров от (N - 1) / 2 до 1. Если квадрат оказывается невозможно вставить в свободную клетку, то его размер уменьшается на 1, и попытка повторяется, если был достигнут размер 0, то шаг заканчивается и глубина рекурсии уменьшается на 1 уровень.

В текущей реализации алгоритма использован ряд решений для оптимизации, чтобы уменьшить требуемое время:

1. Если длина стороны N изначального квадрата четная, то минимальным вариантом будет вариант из 4 квадратов со стороной N / 2, поэтому сразу выводится ответ без использования алгоритма бэктрекинга.

2. Если число N составное и нечетное, то находится минимальный простой делитель числа N, после чего в ответ и битовую матрицу заполения добавляются два квадрата со строной равной отношению N на минимальный простой делитель, умноженное на отношение минимального простого делителя к 2 с округлением к меньшему, а также один квадрат со стороной равной отношению N к минимальному делителю, умноженное на отношение делителя к 2 с округлением к большему. После чего запускается алгоритм бэктрекинга для нахождения недостающих квадратов.
3. Если число N простое, то в левый верхний угол битовой матрицы заполнения добавляется квадрат со стороной $(N + 1) / 2$, а снизу и справа от него квадраты со стороной $(N - 1) / 2$. После чего запускается алгоритм бэктрекинга.
4. На каждом шаге алгоритма находится сторона максимально большого квадрата, который можно вставить. Если эта сторона равна 1 или 2, то все пустые клетки заполняются без алгоритма бэктрекинга.
5. Если на очередном шаге алгоритма количество квадратов стало равным или превысило количество квадратов в минимальном варианте, то обработка этого варианта заканчивается, т. к. он не может быть минимальным.

Сложность алгоритма

Чтобы оценить сложность алгоритма, необходимо оценить сложность каждой функции, требуемой для реализации алгоритма. Обозначим сторону квадрата — N.

Если N — четное число, то ответ выводится без использования циклов и рекусрсий, поэтому сложность по числу операций и памяти — $O(N)$.

Если N — нечетное число, то для нахождения ответа требуется ряд функций, рассмотрим их поотдельности.

Для нахождения минимального простого делителя перебираются числа от 3 до $\sqrt{N}$, следовательно сложность по числу операций — O(N), т. к. данная операция не требует дополнительной памяти, то сложность по памяти — O(1).

Для копирования битовой матрицы заполнения на каждом шаге алгоритма используются два вложенных цикла для создания и заполения двумерного списка размера NxN элементами из переданной битовой матрицы заполнения. Сложность по числу операций и по памяти - O(N^2).

Алгоритм поиска верхней левой свободной клетки использует два вложенных цикла, чтобы перебрать клетки битовой матрицы заполнения. Для перебора используются координаты из диапозона [(N - 1) / 2; N], поэтому сложность по числу операций будет O(N^2). Алгоритм не требует дополнительной памяти, поэтому сложность по памяти — O(1).

Для нахождения стороны максимально большого квадрата, который можно вставить в пустые клетки используется алгоритм с двумя вложенными циклами, которые перебирают координаты x, y в диапозоне [(N - 1) / 2; N]. После нахождения каждой пустой клетки при помощи двух циклов, перебирающих координаты в диапозонах [x, N] и [y, N], следовательно в худшем случае сложность по числу операций будет - O(N^3), т. к. для алгоритма не требуется дополнительная память, то сложность по памяти — O(1).

Алгоритм проверки на то, можно ли вставить в клетку квадрат с выбраным размером стороны состоит из двух вложенных циклов, перебирающих координаты в диапозонах [y, y + $side$] и [x, x + $side$] для первого и второго цикла соответсвенно, где $side$ — сторона квадрата, который

хотим вставить. Поэтому сложность по числу операций — O(*N),* а по памяти — O(1).

Также используются алгоритмы для добавления у удаления квадрата из битовой матрицы заполнения, оба алгоритма используют два вложенных цикла, перебирающих координаты в диапозонах [*y, y + side*] и [*x, x + side*]. Сложность по числу операций — O(N^2), а по памяти — O(1), т. к. дополнительная память не требуется.

На каждом шаге рекурсивного алгоритма бэктрекинга в цикле перебираются все возможные размеры стороны нового квадрата в диапозоне [(*N* - 1) / 2; 1]. После успешной вставки нового квадрата начинается новый шаг алгоритма бэктрекинга с новым перебором размеров стороны. Следовательно количество операций перебора всех вариантов может достигать числа N^N . Т. к. решение на каждом шаге сохраняется частичное решение, но, если оно оказывается неподходящим, оно удаляется, то сложность по памяти будет O($N^N - N^{N-1}$) = O(N^N). Сложность по числу операций связана со всеми функциями, использоваными в реализации алгоритма, поэтому итоговая сложность получается O(*N*) * O(*N*) * O(N^2) * O(N^2) * O(N^3) * O(*N)* * O(N^N) = O(N^N).

Функции и структуры данных

Для хранения данных и их обработки создан класс *Square.*

Класс имеет семь полей: *self.n* — дилна стороны исходного квадрата, *self.k* — минимальное количество квадратов, которыми можно полностью заполнить исходный квадрат, изначально 0, *self.squareList* — список элементов формата [*x, y, w*], где *x, y* — координаты верхнего левого угла, *w* — длина стороны, *self.squareBoolMap* — битовая матрица заполнения, изначачально заполнена 0, *self.minPrimeNumber* — минимальный простой делитель, изначально 1, *self.maxSize* — максимальный размер стороны

квадрата, который будет вставляться в алгоритме, *self.charMap* — список символов, для обозначения разных квадратов в битовой матрице заполнения.

Для вывода конечного и промежуточных результатов, а также для обработки данных в классе реализован ряд методов.

Метод *print(self)* используется для вывода итогового результата. При вызове метод вывод количество квадратов, которое храниться в поле *self.k,* и список координат и длин сторон квадратов, хранящийся в поле *self.squareList.*

Для запуска алгоритма бэктрекинга используется метод *pave(self).* Если длина стороны исходного квадрата — четное число, то сразу сохраняется ответ, рекурсивный алгоритм не вызывается. Если число нечетное, то сначала определяется его минимальный простой делитель, который сохраняется в поле *self.minPrimeNumber*, в зависимости от того простое или составное число в предварительный результат записываются три квадрата определнного размера, после чего вызывается метод рекурсивного алгоритма бэктрекинга.

Для рекурсивного нахождения минимального количества квадратов, которыми можно заполнить исходный квадрат, используется метод *paveRec(self, squareBoolMap, squareList, k, indexMap).* Метод принимает: *squareBoolMap* — битовую матрицу заполнения, *squareList* — список квадратов, *k* — количество квадратов, *indexMap* — индекс символа, которым будет обозначаться новый квадрат. Метод находит и сохраняет в поля класса минимальное количество квадратов, список этих квадратов и битовую матрицу заполнения с этими квадратами. Частичные решения храняться в двумерном списке — битовая матрица заполнения — *squareBoolMap,* и списке — координаты и длины сторон квадратов — *squareList.* На каждом шаге новом шаге рекурсии частичные решения копируются и дальнейшии изменения происходят уже с копиями. В новый вызов рекурсии также передаются копии. Метод рекурсивно вызывает самого себя для нахождения наилучшего результата.

Для определения верхней левой пустой клетки используется метод *searchFreeCoord(self, squareBoolMap)*. Метод принимает *squareBoolMap* — битовую матрицу заполнения, и возвращает либо координаты верхеней левой пустой клетки, либо пару -1, -1, если пустых клеток нет.

Метод *searchSideNewSquare(self, squareBoolMap, newX, newY)* используется для нахождения размера стороны квадрата, который можно вставить по заданным координатам. Метод принимает на вход битовую матрицу заполнения — *squareBoolMap*, координаты x и y — *newX, newY* соответсвенно. Метод возвращает число — длину стороны наибольшего квадрата, который можно вставить.

Для нахождения длины наибольшего квадрата, который можно вставить в незаполенную часть, используется метод *searchMaxSide(self, squareBoolMap)*. Метод принимает битовую матрицу заполнения — *squareBoolMap*, перебирает все клетки битовой матрицы заполнения, находя длину наибольшего квадрата, котоырй можно вставить. Метод возвращает координаты верхнего левого угла наибольшего квадрата и длину его стороны. Если оказывается, что вставить какой-либо квадрат невозможно, то возвращается тройка значений вида -1, -1, 0.

Для проверки того, можно ли вставить квадрат определенного размера по заданным координатам, используется метод *checkSquareCoord(self, squareBoolMap, x, y, n)*. Метод принимает битовую матрицу заполнения — *squareBoolMap*, координаты верхнего левого угла — *x* и *y*, длину стороны квадрата — *n*. Метод возвращает *False*, если вставить квадрат невозможно, в противном случае возвращает *True*.

Для заполнения пустых клеток в битовой матрице заполнения используется метод *paveBoolMap(self, squareBoolMap, x, y, n, i)*, который принимает битовую матрицу заполнения - *squareBoolMap*, координаты левого верхнего угла квадрата, который необходимо вставить в матрицу — *x* и

y, длину стороны квадрата — *n*, индекс символа, которым будет обозначаться квадрат на битовой матрице заполнения. Все клетки, входящие в квадрат, заполняются соответствующим символом. Метод возвращает битовую матрицу заполнения с вставленным квадратом.

Для удаления квадрата из битовой матрицы заполнения используется метод *removePaveBoolMap(self, squareBoolMap, x, y, n)*. Метод принимает на вход битовую матрицу заполнения — *squareBoolMap*, координаты левого верхнего угла квадрата, который необходимо вставить в матрицу — *x* и *y*, длину стороны квадрата — *n*. Метод заменяет символы, входящие в квадрат с верхним левым углом в клетке *x*, *y* и размером *n*, на 0. Метод возвращается битовую матрицу заполнения без квадрата.

Для вывода итоговой битовой матрицы заполнения используется метод *printBoolMap(self)*, который выводит в стандартный поток вывода битовую матрицу заполнения, сохраненную в поле *self.squareBoolMap*.

Метод *printMap(self, squareBoolMap)* используется для вывода промежуточных битовых матриц заполнения. Метод принимает битовую матрицу заполнения *squareBoolMap* и выводит её в стандартный поток вывода.

Исследование

Зависимость времени выполнения от размера квадрата.

Для исследования были проведены замеры времени, требуемого для нахождения решения с минимальным количеством квадратов, для всех квадратов с размером стороны от 2 до 20 включительно. Результат замеров времени представлен на рис. 1.

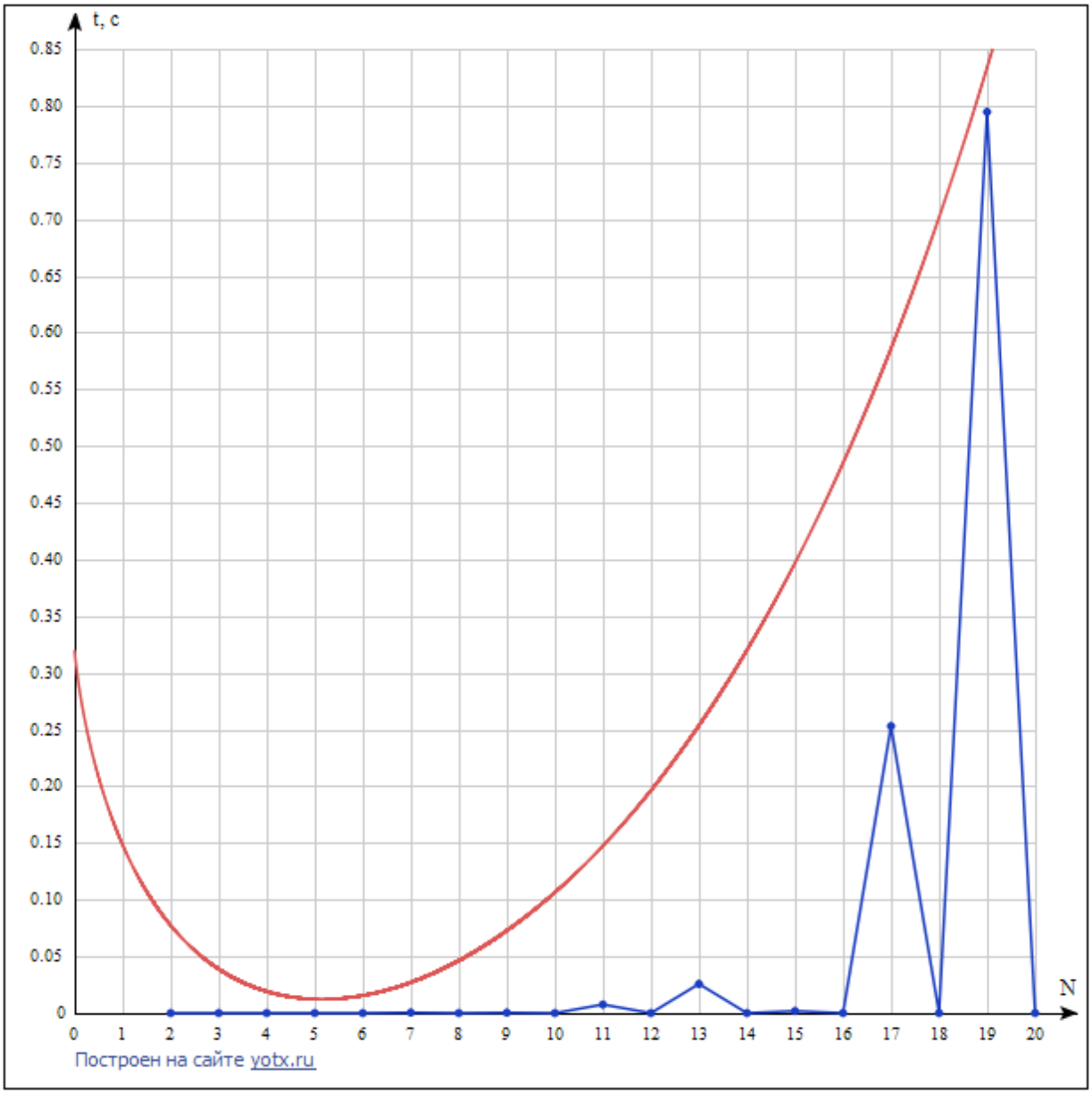


Рисунок 1 — Зависимость времени от длины стороны. (Синяя — зависимость времени от длины стороны, красная — верхняя ассимптотическая граница)

По рисунку видно, что для четных чисел требуемое время растет незначительно. Для $N = 2$ требуется 0,00001 секунды, а для $N = 20$ — 0,00004 секунды. Для нечетных чисел скорость роста требуемого времени зависит от того, простое число или составное. Для составных чисел 9 и 15 требуется 0,00033 и 0,00182 секунды соотвественно, следовательно количество требуемого времени по сравнению с четными числами растет быстрее. Для

простых чисел требуемое время учеличивается быстрее чем для других чисел, для N = 3 — 0,00003 секунда, а для N = 19 — 0,79441 секунды. Следовательно с увеличением размера исходного квадрата, увеличивается требуемое время. Полученные результаты подтверждают теорретическую оценку сложности - O(N^N).

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

Выводы.

В ходе работы был рассмотрен алгоритм бэктрекинга. Была написана программа, использующая рекурсивную реализацию алгоритма бэктрекинга для нахождения оптимального варианта разбиения квадрата на квадраты меньшего размера. Сложность алгоритма по числу операций — O(N^N), по памяти — O(N^N).

Было проведено исследование зависимости времени выполнения от размера квадрата. Полученные результаты подтвердили теорретическую оценку сложности — O(N^N).

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
import time

class Square:
    def __init__(self, n):
        self.n = n      # Длина стороны
           self.k = 0        # Кол-во квадратов, которыми можно
замостить
        self.squareList = []
          self.squareBoolMap = [[0 for a in range(n)] for b in
range(n)]      # Карта заполнения
        self.minPrimeNumber = 1
        self.maxSize = int((n-1)/2)
         self.charMap = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
'I', 'J', 'K', 'L', 'M', 'N', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
'X']

    def print(self):    # Вывод результатов
        print(self.k)
        for element in self.squareList:
            print(*element, sep=' ')

       def  pave(self):         #  Нахождение минимального числа
квадратов, которыми можно замостить квадрат со стороной n
        time_start = time.perf_counter()
        if self.n % 2 == 0:     # Если сторона четной длины
            self.k = 4
            self.squareList.append([1, 1, int(self.n / 2)])
                                                self.squareBoolMap   =
self.paveBoolMap(self.squareBoolMap, 0, 0,  int(self.n / 2), 0)

                    self.squareList.append([1, 1 + int(self.n / 2),
int(self.n / 2)])
                                                self.squareBoolMap   =
self.paveBoolMap(self.squareBoolMap, 0, int(self.n / 2),  int(self.n /
2), 1)

                    self.squareList.append([1 + int(self.n / 2), 1,
int(self.n / 2)])
                                                self.squareBoolMap   =
self.paveBoolMap(self.squareBoolMap, int(self.n / 2), 0,  int(self.n /
2), 2)

                   self.squareList.append([1 + int(self.n / 2), 1 +
int(self.n / 2), int(self.n / 2)])
                                                self.squareBoolMap   =
self.paveBoolMap(self.squareBoolMap, int(self.n / 2), int(self.n / 2),
int(self.n / 2), 3)

        else:   # Если сторона нечетной длины
```

```
                    for i in range(3, int(self.n ** (1/2) + 1)):     #
Нахождение минимального просто делителя
                        if self.n % i == 0:
                            self.minPrimeNumber = i
                            break

                if self.minPrimeNumber != 1:     # Составные числа
                    self.maxSize = int(self.n / self.minPrimeNumber)
                    coeff = int(self.minPrimeNumber / 2)

                                            self.squareList.append([1,  1,
self.maxSize*(coeff+1)])
                                                    self.squareBoolMap   =
self.paveBoolMap(self.squareBoolMap, 0, 0, self.maxSize*(coeff+1), 0)

                                           self.squareList.append([1,  1  +
self.maxSize*(coeff+1), self.maxSize])
                                                    self.squareBoolMap   =
self.paveBoolMap(self.squareBoolMap,     0,     self.maxSize*(coeff+1),
self.maxSize*coeff, 1)

                                             self.squareList.append([1  +
self.maxSize*(coeff+1), 1, self.maxSize])
                                                    self.squareBoolMap   =
self.paveBoolMap(self.squareBoolMap,     self.maxSize*(coeff+1),     0,
self.maxSize*coeff, 2)

                        self.paveRec(self.squareBoolMap, self.squareList,
3, 3)

                else:   # Простые числа
                    self.squareList.append([1, 1, self.maxSize + 1])
                                                    self.squareBoolMap   =
self.paveBoolMap(self.squareBoolMap, 0, 0, self.maxSize + 1, 0)

                            self.squareList.append([1, self.maxSize + 2,
self.maxSize])
                                                    self.squareBoolMap   =
self.paveBoolMap(self.squareBoolMap,     0,     self.maxSize     +     1,
self.maxSize, 1)

                            self.squareList.append([2 + self.maxSize, 1,
self.maxSize])
                                                    self.squareBoolMap   =
self.paveBoolMap(self.squareBoolMap,     self.maxSize     +     1,     0,
self.maxSize, 2)

                        self.paveRec(self.squareBoolMap, self.squareList,
3, 3)

                       print('Time:{0:.5f}'.format(time.perf_counter()  -
time_start))
            #return time.perf_counter() - time_start

        def paveRec(self, squareBoolMap, squareList, k, indexMap):
```

```
            if k >= self.k and self.k != 0:
                    print('Количество квадратов превысило минимальное
количество')    # Промежуточный вывод
                  print('Все квадраты со стороной длины 1 и 2 удалены\
n')
                return 0

            bufBoolMap = [[a for a in b] for b in squareBoolMap]#
Копирование данных
            bufList = [element for element in squareList]

            x, y = self.searchFreeCoord(bufBoolMap)

            if x < 0 and y < 0:# Если нельзя вставить ни один квадрат
                if self.k > k or self.k == 0:
                      print('Решение сохранено как новое минимальное\
n')    # Промежуточный вывод
                         print('Все квадраты со стороной длины 1 и 2
удалены\n')
                    self.k = k
                    self.squareList = bufList
                    self.squareBoolMap = bufBoolMap
                else:
                    print('Количество квадратов превысило минимальное
количество')    # Промежуточный вывод
                         print('Все квадраты со стороной длины 1 и 2
удалены\n')
                return 0

            bufX, bufY, maxSize = self.searchMaxSide(bufBoolMap)
            while maxSize == 2:# Если максимальный квадрат с размером
2
                k += 1
                bufList.append([bufX + 1, bufY + 1, 2])
                self.paveBoolMap(bufBoolMap, bufX, bufY, 2, indexMap)
                 print('x:', bufX+1, 'y:', bufY+1, 'w:', 2, '\nk =',
k)  # Промежуточный вывод
                self.printMap(bufBoolMap)  # Промежуточный вывод
                indexMap += 1
                bufX, bufY, maxSize = self.searchMaxSide(bufBoolMap)
                if k >= self.k and self.k != 0:
                    print('Количество квадратов превысило минимальное
количество')    # Промежуточный вывод
                         print('Все квадраты со стороной длины 1 и 2
удалены\n')
                    return 0

            if maxSize <= 1:    # Если максимальный размер - 1
                for bufY in range(self.maxSize, self.n):
                    for bufX in range(self.maxSize, self.n):
                        if not bufBoolMap[bufY][bufX]:
                            k += 1
                            bufList.append([bufX+1, bufY+1, 1])
                                        bufBoolMap[bufY][bufX]  =
self.charMap[indexMap]
```

```
                                print('x:', bufX+1, 'y:', bufY+1, 'w:',
1, '\nk =', k)  # Промежуточный вывод
                                    self.printMap(bufBoolMap)    #
Промежуточный вывод
                            indexMap += 1
                            if k >= self.k and self.k != 0:
                                print('Количество квадратов превысило
минимальное количество')    # Промежуточный вывод
                                print('Все квадраты со стороной длины
1 и 2 удалены\n')
                                return 0
                if k < self.k or self.k == 0:
                     print('Решение сохранено как новое минимальное\
n')    # Промежуточный вывод
                        print('Все квадраты со стороной длины 1 и 2
удалены\n')
                    self.k = k
                    self.squareList = bufList
                    self.squareBoolMap = bufBoolMap
                else:
                    print('Количество квадратов превысило минимальное
количество')    # Промежуточный вывод
                        print('Все квадраты со стороной длины 1 и 2
удалены\n')
                return 0

            for m in range(self.maxSize, 0, -1):     #  Перебор  всех
размеров для нового квадрата
                if self.checkSquareCoord(squareBoolMap, x, y, m):
                    self.paveBoolMap(bufBoolMap, x, y, m, indexMap)
                        print('Вставлен квадрат x:', x+1, 'y:', y+1,
'w:', m, '\nk =', k)  # Промежуточный вывод
                    self.printMap(bufBoolMap)  # Промежуточный вывод
                    bufList.append([x+1, y+1, m])
                              self.paveRec(bufBoolMap,  bufList,  k+1,
indexMap+1)
                    self.removePaveBoolMap(bufBoolMap, x, y, m)
                     print('Удален квадрат x:', x+1, 'y:', y+1, 'w:',
m)
                    self.printMap(bufBoolMap)
                    bufList.pop()

            return 0

          def  searchFreeCoord(self,  squareBoolMap):     # Нахождение
верхнего левого угла свободного от квадратов
            for y in range(self.maxSize, self.n):
                for x in range(self.maxSize, self.n):
                    if not squareBoolMap[y][x]:
                        return x, y
            return -1, -1

        def searchSideNewSquare(self, squareBoolMap, newX, newY):   #
Нахождение стороны максимально большого квадрата,
```

```
            newSideX = newSideY = 0                                    #
который можно вставить по заданным координатам
            for x in range(newX, self.n):
                if not squareBoolMap[newY][x]:
                    newSideX += 1
                else:
                    break
            for y in range(newY, self.n):
                if not squareBoolMap[y][newX]:
                    newSideY += 1
                else:
                    break
            if newSideX <= newSideY:
                return newSideX
            return newSideY

        def searchMaxSide(self, squareBoolMap):        # Нахождение
стороны максимально большого квадрата, который можно
            bufX = bufY = 0                            # вставить в
свободную часть
            newSide = 0
            for newY in range(self.maxSize, self.n):
                for newX in range(self.maxSize, self.n):
                                                       bufNewSide   =
self.searchSideNewSquare(squareBoolMap, newX, newY)
                    if bufNewSide > newSide:
                        newSide = bufNewSide
                        bufY = newY
                        bufX = newX
                    if bufY + newSide >= self.n:
                        if newSide == 0:
                            return -1, -1, 0
                        return bufX, bufY, newSide
            if newSide == 0:
                return -1, -1, 0
            return bufX, bufY, newSide

        def checkSquareCoord(self, squareBoolMap, x, y, n):      #
Можно ли вставить квадрат с заданной
            if x + n > self.n or y + n > self.n:               #
стороной на координаты x y
                return False
            for bufY in range(y, y+n):
                side = 0
                for bufX in range(x, x+n):
                    if not squareBoolMap[bufY][bufX]:
                        side += 1
                if side < n:
                    return False
            return True

        def paveBoolMap(self, squareBoolMap, x, y, n, i):        #
Заполнение пустых клеток на карте
            for bufY in range(y, y+n):
                for bufX in range(x, x+n):
```

```
                    squareBoolMap[bufY][bufX] = self.charMap[i]
            return squareBoolMap

        def removePaveBoolMap(self, squareBoolMap, x, y, n):     #
Удаление квадрата на карте по координатам и стороне
            for bufY in range(y, y+n):
                for bufX in range(x, x+n):
                    squareBoolMap[bufY][bufX] = 0
            return squareBoolMap

        def printBoolMap(self):     # Вывод карты заполнения
            for y in range(self.n):
                for x in range(self.n):
                    print(self.squareBoolMap[y][x], end='')
                print()

        def printMap(self, squareBoolMap):
            for y in range(self.n):
                for x in range(self.n):
                    print(squareBoolMap[y][x], end='')
                print()
            print()



    n = int(input())
    if n < 2 or n > 20:
        print('Неверный ввод')
        exit()
    square = Square(n)
    square.pave()
    square.print()
    print('\nИтоговая матрица заполнения:')
    square.printBoolMap()

    '''squareFile = open("research.txt", "w")   # Исследование

    for a in range(2, 21):
            square = Square(a)
            atime = square.pave()
                    squareFile.write("("  +  str(a)  +  ";  {0:.5f})\
n".format(atime))'''
```

ПРИЛОЖЕНИЕ Б

ТЕСТИРОВАНИЕ

Таблица Б.1 - Примеры тестовых случаев

№ п/п	Входные данные	Выходные данные	Комментарии
1.	2	4 1 1 1 1 2 1 2 1 1 2 2 1	
2.	3	6 1 1 2 1 3 1 3 1 1 3 2 1 2 3 1 3 3 1	
3.	5	8 1 1 3 1 4 2 4 1 2 4 3 2 3 4 1 3 5 1 4 5 1 5 5 1	
4.	7	9 1 1 4 1 5 3 5 1 3 5 4 2 4 6 2 6 6 2 7 4 1	

		4 5 1 7 5 1	
5.	10	4 1 1 5 1 6 5 6 1 5 6 6 5	Т.к. 10 — четное число, то его можно разбить на 4 квадрата
6.	13	11 1 1 7 1 8 6 8 1 6 8 7 2 10 7 4 7 8 1 7 9 3 10 11 1 11 11 3 7 12 2 9 12 2	
7.	15	6 1 1 10 1 11 5 11 1 5 11 6 5 6 11 5 11 11 5	Т.к. наименьший простой делитель — 3, то разбиение соответсвует разбиению для 3, увеличенному в 5 раз
8.	19	13 1 1 10 1 11 9 11 1 9 11 10 3 14 10 6 10 11 1	

		10 12 1 10 13 4 14 16 1 15 16 1 16 16 4 10 17 3 13 17 3	